NASA Contractor Report 187601
ICASE Report No. 91-55

ICASE

RECTILINEAR PARTITIONING OF IRREGULAR DATA PARALLEL COMPUTATIONS

David M. Nicol

Contract No. NAS1-18605
July 1991

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Rectilinear Partitioning of Irregular Data Parallel Computations

David M. Nicol*

College of William and Mary

Abstract

This paper describes new mapping algorithms for domain-oriented data-parallel computations, where the workload is distributed irregularly throughout the domain, but exhibits localized communication patterns. We consider the problem of partitioning the domain for parallel processing in such a way that the workload on the most heavily loaded processor is minimized, subject to the constraint that the partition be perfectly rectilinear. Rectilinear partitions are useful on architectures that have a fast local mesh network and a relatively slower global network; these partitions heuristically attempt to maximize the fraction of communication carried by the local network. This paper provides an improved algorithm for finding the optimal partition in one dimension, new algorithms for partitioning in two dimensions, and shows that optimal partitioning in three dimensions is NP-complete. We discuss our application of these algorithms to real problems.

*This research was supported in part by the Army Avionics Research and Development Activity through NASA grant NAG-1-787, in part by NASA grant NAG-1-1132, and in part by NSF Grant ASC 8819373.

1 Introduction

One of the most important problems one must solve in order to use parallel computers is the mapping of the workload onto the architecture. This problem has attracted a great deal of attention in the literature, leading to a number of problem formulations. One often views the computation in terms of a graph, where nodes represent computations and edges represent communication; for example, see [2]. Mapping means assigning each node to a processor; this is equivalent to partitioning the nodes of the graph, with the tacit understanding that nodes in a common partition set are assigned to the same processor. We will use the terms interchangeably. A common mapping problem formulation views the architecture as a graph whose nodes are processors and whose edges identify processors able to communicate directly. The *dilation* of a computation graph edge (u, v) is the minimum distance (in the processor graph) between the processors to which u and v are respectively assigned. The dilation of the graph itself is the maximum dilation among all computation graph edges. Dilation is a measure of how well the mapping preserves locality between nodes in the mapped computation graph. Results concerning the minimization of dilation can be found in [4, 9, 16, 21], and their references.

Another formulation (the one we study) directly models execution time of a data parallel computation as a function of the chosen mapping, and attempts to find a mapping that minimizes the execution time. Workload may again be represented as a graph, with edges representing data communication, e.g., the stencils used in some partial differential equation solvers [18]. In its simplest form each node is assumed to have unit execution weight; more general forms permit nodes to have individual weights. Nodes are mapped to processors in such a way that each processor's sum of node weights is approximately the same, for example, see [1, 3, 19]. A rigorous treatment of partitioning three dimensional finite-difference and finite-element domains is found in [23]; unlike our treatment here, the shape of the subdomains are not a consideration. Minimization of communication costs subject to load-balancing constraints is considered in [6]; other formulations use simulated annealing or neural networks to minimize an "energy" function that heuristically quantifies the cost of the partition [7]. Other interesting formulations consider mapping highly structured computations onto pipelined multiprocessors[14], and mapping systolic algorithms onto hypercubes [10].

This paper considers the *Rectilinear Partitioning Problem* (RPP): find an optimal *rectilinear* partition of a domain containing irregularly weighted workload. One may view the workload as being concentrated at discrete coordinates within the domain; alternatively one may represent the domain's workload in a workload matrix, each of whose elements represents all the workload within a rectangular fixed-sized region of the domain. A domain with irregularly distributed discrete workload can always be transformed into a workload matrix; our problem formulation will thus be stated in terms of the matrix view. A rectilinear partition of a workload matrix requires each partition element to be an appropriately dimensioned "rectangle" whose dimensions exactly match that of each neighbor at each face. "Rectangles" in one dimension are intervals; a "rectangle" in three dimensions is a rectangular solid. The cost of a rectangle is the sum of all workload within its boundaries; the cost of the partition is the maximum cost among all its rectangles. The idea

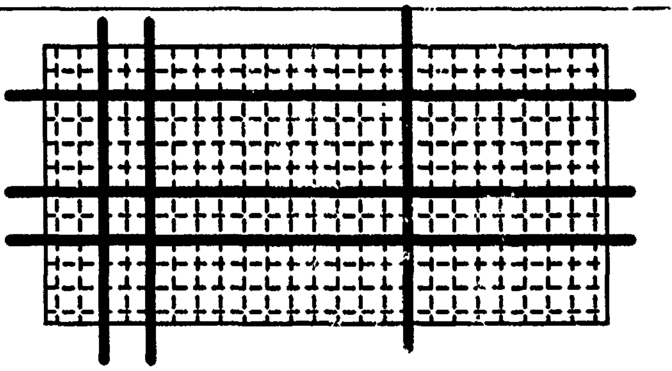


Figure 1: Two dimensional 4 × 4 rectilinear partition of a workload matrix representing a two-dimensional domain

is to let the weight of a rectangle be the time required to execute its implicitly assigned workload on a processor. The maximum such defines the computation's finishing time (or its inverse defines the processing rate) when each rectangle is assigned to a different processor. Figure 1 illustrates a rectilinear partition in two dimensions.

RPP arises when executing physically-oriented numerical computations on certain types of mesh-connected multiprocessors. For example, some computations are based on grids that discretize a one, two or three dimensional field for numerical solution. An $n \times m$ workload matrix can be constructed by pre-aggregating adjacent grid points to create a rectangular structure; alternatively, one may chose n and m so large that at most one grid point is represented in one cell of the $n \times m$ domain. RPP is motivated by parallel architectures that support very fast "local" communication over a mesh network, and significantly slower "global" communication. The Connection Machine provides an example: the speed differential between communication using the local network and the global router is roughly a factor of six on problems with regular communication patterns [22]. It can be worse if the global network suffers significant contention. Since communication requirements in domain-oriented computations are often localized in space, rectilinear partitions will tend to maximize the volume of communication which can use the fast mesh network.

It is possible to partition a domain with irregularly distributed discrete workload into quadrilaterals whose faces match exactly, as do rectilinear partitions. Such partitions will have the desirable locality of communication properties we seek. However, rectilinear partitions have the advantage of being expressed simply. One benefit is that one can always *compute* the processor id of a point (x, y) with whom wishes to communicate: a binary search on the list of cuts in the X dimension establishes the X processor coordinate of (x, y)'s processor, another search establishes the Y processor coordinate. Simplicity of expression also implies simplicity of construction. There is some advantage to choosing $N + M - 2$ cuts instead of choosing $(N - 1) \times (M - 1)$ cuts. Finally, the mathematical regularity of rectilinear partitions make them interesting objects in their own right.

We consider partitioning in one, two, and three dimensions. It should be noted that a three dimensional domain can always be partitioned for a two or one dimensional processor array; likewise, a two dimensional domain can be partitioned for a one dimensional processor array. Thus, the partitioning dimension describes the communication topology of the target architecture. A distinction can be made between processor meshes that directly connect diagonally adjacent processors, and those that don't. The algorithms we develop here are primarily concerned with the latter. They may be used on more fully connected meshes, but do not attempt to take advantage of the extra connectivity.

RPP is a challenging problem, as it is similar to certain NP-complete problems, but is also similar to problems with polynomial complexity. It is already known that the one-dimensional problem can be solved in polynomial time [3, 5, 17]. Our first result is to improve upon the best published 1D algorithm to date, for the case when the computation's size greatly exceeds the number of processors. Next we consider RPP in two dimensions. We show that if the partition in one dimension (say x) is fixed, then the optimal partitioning in the other dimension can be found in polynomial time. This result has at least two applications. First, it can be used to find the optimal 2D rectilinear partition; one simply generates all partitions in one dimension, and finds, for each, the optimal partitioning in the other. While this procedure is correct, it has unreasonably high complexity. For this reason we develop a 2D *iterative refinement* heuristic based on our ability to find conditionally optimal partitions. During each iteration, one finds the optimal partitioning in one dimension, given a fixed partition in the other. The next iteration uses the solution just found as the fixed partition, and optimally solves in the other dimension. One then iterates until the partition stops changing. The procedure is guaranteed to converge monotonically to a local minimum in the solution space. We discuss application of this technique to two-dimensional problems arising in fluid dynamics calculations, and compare the quality of solutions produced by the heuristic with solutions produced by algorithms having fewer restrictions on the partitioning, e.g., binary dissection. We find that rectilinear partitions can achieve better performance than the other methods, especially when the grid edges are oriented in two orthogonal directions, or when global communication is an order of magnitude slower than local communication. Our last contribution is to show that the problem of finding an optimal rectilinear partition in three dimensions is NP-complete.

The remainder of this paper is organized as follows. In Section §2 we introduce some notation and develop the cost function we wish to minimize. In Section §3 we give an improved solution to RPP in one dimension. Section §4 examines RPP in two dimensions, and Section §5 proves the NP-completeness of finding optimal three-dimensional partitions. Section §6 summarizes our results.

2 Preliminaries

This section introduces some notation used throughout the paper. The discussion to follow speaks of the two dimensional partitioning problem; the extension to three dimensions is immediate, and the projection to one dimension simply involves dropping notational dependence on indices in one dimension.

We define the partitioning problem as follows. Consider an $n \times m$ *load matrix* $\{L_{i,j}\}$, where each entry $L_{i,j} \geq 0$ represents the cost of executing some workload. For example, we might create a load matrix from a discretized domain by prepartitioning the domain into many rectangular *work pieces*, and assign load value $L_{i,j}$ to work piece $w_{i,j}$, based on the number of grid points and edges defined within $w_{i,j}$. In the limit, we may prepartition the domain so finely that a work piece represents at most one grid point and its edges.

Our problem is to partition the load matrix for execution on an $N \times M$ array of processors, as follows. A partition is defined to be a pair of vectors (R, C), where R is an ordered set of row indices $R = (r_0, r_1, \ldots, r_N)$, C is an ordered set of column indices $C = (c_0, c_1, \ldots, c_M)$, and we understand that $r_0 = c_0 = 0$, $r_M = m$, and $c_N = n$. Given (R, C), the *execution load* on processor $P_{i,j}$ is the sum of the weights of all the work pieces $w_{x,y}$ with $r_{i-1} < x \leq r_i$, and $c_{j-1} < y \leq c_j$. This is given by

$$X_{i,j}(R,C) = \sum_{x=r_{i-1}+1}^{r_i} \sum_{y=c_{j-1}+1}^{c_j} L_{x,y}.$$

We take the overall cost of the partition to be the maximal execution load assigned to any processor:

$$\pi(R,C) = \max_{\text{all } i \text{ and } j} \{X_{i,j}(R,C)\}.$$

This cost is known as the *bottleneck* value for the partition. Our object is to find partition vectors R and C that minimize the bottleneck value.

We have chosen not to include explicit communication costs in this model. This is a largely practical decision. The data communication inherent in a computational problem tends to be proportional to execution costs. This means that by balancing the execution load we will have greatly balanced the communication load also, at least if the bandwidth of the network is high enough for us to ignore contention. It is also true that the execution weights $L_{i,j}$ are only estimates to begin with; it seems unlikely that a more complicated model will find significantly better partitions. Finally, we are assuming that rectilinear partitioning is desirable because local communication is much cheaper than global communication. If we can ensure that the partition supports local communication we will have gone a long way towards minimizing communication overhead. Our empirical study discussed in §4.5 bears out this intuition.

3 One Dimensional Partitioning

RPP in one dimension has been extensively studied as the *chains-on-chains* partitioning problem [3, 5, 11, 12, 17]: we are given a linear sequence of work pieces (called modules), and wish to partition the sequence for execution on a linear array of processors. Until recently, the best published algorithm found the optimal partitioning in $O(Mm \log m)$ time, where M is the number of processors and m is the number of modules. This solution and those developed in [11] and [17] all involve repeatedly calling a *probe* function. A recently discovered dynamic programming formulation [5] reduces the complexity further to $O(Mm)$. The solution we present has a complexity of $O(m+(M\log m)^2)$, which is better than $O(Mm)$ when $M = O(m/\log^2 m)$. This solution is also based on a probe function, which we now discuss in more detail.

In one dimension we are to partition a chain of modules $w_1,\ldots,w_m$ with weights $L_1,\ldots,L_m$ into M contiguous subchains. We use a function **probe**, which accepts a *bottleneck constraint* W and determines whether any partition exists with bottleneck value W', where $W' \leq W$. Candidate constraints all have the form $W_{i,j} = \sum_{k=i}^{j} L_k$, because we know that the optimal cost is defined by the load we place on some processor. If we precompute the m sums $W_{1,j}$ $(j = 1,\ldots,m)$, then any candidate value $W_{i,j}$ can be generated in $O(1)$ time using the relationship $W_{i,j} = W_{1,j} - W_{1,i-1}$.

Given bottleneck constraint W, **probe** attempts to construct a partition with a bottleneck value no greater than W. The first processor must contain the first module; **probe** finds the largest index i_1 such that $W_{1,i_1} \leq W$, and assigns modules 1 through i_1 to the first processor. Because the sums $W_{1,j}$ increase in j, i_1 can be found with a binary search. It follows that the first module in the second processor is w_{i_1+1}. **probe** then loads the second processor with the longest subchain beginning with w_{i_1+1} that does not overload the processor. This process continues until either all modules have been assigned, or the supply of processors is exhausted. In the former case we know that a feasible partition with weight no greater than W exists. In the latter case we know that this greedy approach does not produce a feasible partition. However, it has been shown (and indeed is quite straightforward to see) that the greedy approach will find a solution with cost no greater than W if any solution exists with cost no greater than W. Since the loading of each processor requires only a binary search, the cost of one **probe** call is $O(M\log m)$.

All solutions based on **probe** search the space of bottleneck constraints for the minimal one, say W_{opt}, such that **probe**(W_{opt}) = **true**. The partition **probe** generates given bottleneck constraint W_{opt} is optimal. The solution in [17] examines no more than $4m$ candidate constraints, which gives the 1D partitioning problem an overall time complexity of $O(Mm\log m)$. As argued by Iqbal[11], another easy way to probe is to compute the sum of all workload in the domain, say Z, and choose a discretization length ϵ. One may then conceptually discretize the interval $[0,Z]$ into Z/ϵ constraints, and use a binary search to find the minimum feasible constraint. This approach has a complexity of $O(M\log(Z/\epsilon)\log m)$, although the cost of a partition it finds is guaranteed only to be within ϵ of optimal. The only disadvantage of this method occurs when $\log(Z/\epsilon)$ is large relative to m, in which case one may choose to search more cleverly. Towards this end we next develop the paper's first contribution, a searching technique that finds the optimal partition after only $O(M\log m)$ **probe** calls[1].

Let W_{opt} be the minimal constraint for which **probe** returns value **true**. The new search strategy exploits the following structure of an optimal solution constructed by **probe**. Suppose processor F is the first processor assigned a load whose weight is exactly W_{opt}. The loads on all processors 1 through $i-1$ must be strictly less than W_{opt}, and hence their loads are not feasible

[1] This result was originally observed by Iqbal (private communication). We present an independently discovered proof (and algorithm) which easily extends to a 2D problem.

bottleneck constraints. However, the greedy construction process ensures that the load on each processor up to F is as large as possible. For example, processor 1 is loaded with the longest subchain, beginning with w_1, whose weight does not exceed W_{opt}. For any $W' < W_{opt}$ we have **probe**(W') = **false**; let i_1 be the largest index such that **probe**(W_{1,i_1}) = **false**. Consider the relationships

$$W_{1,i_1} < W_{opt} \leq W_{1,i_1+1}.$$

If $W_{opt} < W_{1,i_1+1}$ (i.e., if $1 \leq F$) then modules 1 through i_1 will be assigned to processor 1, otherwise module $i_1 + 1$ will also be assigned to 1. Supposing that $1 < F$, the subchain assigned to processor 2 begins with module w_{i_1+1}; define i_2 to be the largest index for which **probe**(W_{i_1+1,i_2}) = **false**. Under the greedy assignment, processor 2's last module is either w_{i_2} or w_{i_2+1}, depending on whether $F = 2$. We may carry out this process for each processor: given i_j, i_{j+1} is the largest index for which **probe**$(W_{i_j+1,i_{j+1}})$ = **false**. For each i_j define $\omega_j = W_{i_{j-1}+1,i_j+1}$. From the discussion above it is apparent that when $j \leq F$, i_j is the first module assigned to processor j under the optimal greedy partition. ω_j is the smallest feasible constraint arising from any subchain beginning with module w_{i_j}. Therefore, $W_{opt} = \omega_F$. Furthermore, each ω_j for $j > F$ is a feasible constraint, and hence must be at least as large as W_{opt}. This proves the following lemma.

Lemma 1 *Let W_{opt} be the minimal feasible bottleneck weight. Then* $W_{opt} = \min_{1 \leq j \leq M}\{\omega_j\}$.

An important point is that the definitions of the i_j's and ω_j's in no way depend on knowledge of either F or W_{opt}. We may discover W_{opt} by generating each constraint ω_i, and choosing the least. In order to find i_1 (and hence ω_1) we need to search the space of all weights having the form $W_{1,j}$. As we have already seen, this space can be searched with only $O(\log m)$ calls to **probe**. Each **probe** call costs $O(M \log m)$; the cost of finding ω_1 is thus $O(M \log^2 m)$. Similarly, given i_1, we find i_2 using a binary search over all weights of the form $W_{i_1+1,j}$, and so on. As there are M such ω_j's to compute, the overall cost of the computation is $O(m + (M \log m)^2)$, where an obligatory $O(m)$ cost is added to account for preprocessing costs. This complexity is better than $O(Mm)$ whenever $M = O(m/\log^2 m)$, showing that the strategy is most useful when there are many modules to be processed relative to processors. This is exactly the situation we face when partitioning large numerical problems. One of the more useful applications of the new algorithm will be as part of an approach for solving two dimensional problems, our next topic.

4 Two Dimensional Partitioning

Next we turn to partitioning in two dimensions. Our discussion has three parts. First we provide some contrast by discussing a closely related 2D partitioning problem which is NP-complete. We then return to our original 2D problem, and describe an algorithm that takes a given fixed column (alternately, row) partition, and finds the optimal partitioning of the rows (alternately, columns) in polynomial time. This result can be used to find an optimal 2D partition, albeit with exponential complexity when N and M are problem parameters.. We describe a heuristic with polynomial-time complexity that finds a local minimum in the solution space. Finally, we discuss our experience with this algorithm on large irregular grids typical of those used to solve fluid flow problems.

4.1 MLAA Problem

Consider a two-dimensional $n \times m$ load matrix representing an n-stage computation, as follows. Each column represents some module, the weight of $w_{i,j}$ represents the computational requirement of module j during "stage" i. The columns are to be partitioned into contiguous groups and mapped onto a linear array of processors. In this respect the problem is one-dimensional; however, the objective function is based on both matrix dimensions, as we will see. We assume that the computation requires global synchronization between stages. The same partitioning of modules is applied to all stages. Thus, a partitioning that is good for one stage may create imbalance in another. The execution time of the ith stage is taken to be that of the most heavily loaded processor during the ith stage, the stage's *bottleneck* value. The overall execution time is then the sum of bottleneck values from all stages. The problem of finding the optimal partitioning of columns is known as the *Multistage Linear Array Assignment* (MLAA) problem[13]. The MLAA problem has been shown to be NP-complete. Solutions with polynomial complexity are known if the number of stages is constant.

The MLAA problem is an interesting point of reference for the two-dimensional partitioning problem, for, by changing the objective function slightly, we obtain a problem related to two-dimensional partitioning that has low polynomial complexity. Suppose we seek a partitioning that minimizes the *maximum* of the stage bottleneck weights, rather than their sum. This problem is equivalent to that of finding the optimal two-dimensional rectilinear partitioning, conditioned on the row (alternatively, column) partitioning being fixed. For example, suppose that row partition R is given for a two dimensional load matrix. We know then that all work pieces lying in a given workload column y between workload row indexes $r_{i-1} + 1$ and r_i will be assigned to the same processor, in the i^{th} row of processors. We may therefore aggregate them into a single super-piece with weight

$$A_{i,y} = \sum_{x=r_{i-1}+1}^{r_i} w_{x,y}.$$

This aggregation creates an $N \times m$ weight matrix A. Any subsequent partitioning of the columns into M contiguous groups completes a rectilinear partitioning. Like the MLAA problem we can compute the weight of the most heavily loaded processor in each row, and call this the row's bottleneck weight. The maximum bottleneck weight is then the maximum execution weight among all processors. However, unlike the MLAA problem, the optimal column partition can be found quickly, as we now show.

4.2 Optimal Conditional Partitioning

The heart of all our 2D partitioning algorithms is an ability to optimally partition in one dimension, given a fixed partition in the other. Suppose a row partition R is given. As described in the previous subsection, we can aggregate work pieces forced (by R) to reside on a common processor into super-pieces, thereby creating an $N \times m$ load matrix $\{A_{i,j}\}$. This matrix can be viewed as N one dimensional chains; a common partitioning of their columns will produce a 2D rectilinear partition.

The problem of finding an optimal column partition can be approached through a minor modification to the 1D **probe** function. Given bottleneck constraint W, we find the largest index c_1 such that

$$\sum_{1 \leq j \leq c_1} A_{i,j} \leq W \qquad \text{for all chains } i.$$

This is accomplished with N binary searches, one per chain, each of which finds the longest subchain whose weight is no greater than W. c_1 is the length of the subchain with fewest modules. Like the 1D probe, this one greedily makes c_1 as large as possible without violating the load constraint in any chain. Workload columns 1 through c_1 are assigned to the processors in column 1 of the processor array. The procedure is repeated, assigning columns $c_1 + 1$ through c_2 to processor column 2, and so on. It is easily proven by induction on M that this procedure will find a partition with cost no greater than W, if one exists. The cost of calling **probe** is $O(NM \log m)$, provided we have precomputed the partial sums of all N chains (a $O(nm)$ startup cost).

We will later exploit a useful, self-evident property of partitions constructed by this procedure.

Lemma 2 *Let W be a feasible bottleneck constraint, and let a row partition be given. Let $C = (c_0, c_1, c_2, \ldots, c_M)$ be the greedy column partition constructed using W, and let $C' = (c'_0, c'_1, c'_2, \ldots, c'_M)$ be any other column partition that gives cost W. Then for all $i = 0, 1, 2, \ldots, M$, $c_i \geq c'_i$.*

The same improved searching strategy as was developed for the 1D problem can be applied here. The argument for Lemma 1 does not depend on the partitioning of a single chain; the key insight driving the proof is recognition of the structure of the optimal greedily constructed partition. The same insight applies to this problem, with slightly expanded notation. For all column indices $i < j$ and row index k, let $W_{i,j,k} = \sum_{t=i}^{j} A_{k,t}$. We define $i_0 = 0$, and for $j = 1, \ldots, M$ define i_j to be the largest index such that

$$\textbf{probe}(\{W_{i_{j-1}+1, i_j, k}\}) = \textbf{false} \qquad \text{for all } k = 1, 2, \ldots, N, \tag{1}$$

and define

$$\omega_j = \min_{1 \leq k \leq N} \{W_{i_{j-1}+1, i_j+1, k} \mid \textbf{probe}(W_{i_{j-1}+1, i_j+1, k}) = \textbf{true}\}. \tag{2}$$

These new definitions correspond to the old ones in the obvious way. Suppose the minimum feasible bottleneck constraint is W_{opt}, and let F be the column processor index of the first column where a processor achieves weight W_{opt}; suppose $F > 1$. To chose i_1 we examine each workload row, and for each find the endpoint of the longest subchain whose weight is strictly less than W_{opt}. We then define i_1 to be the smallest among these, say for row r. Since $W_{1,i_1+1,r} > W_{opt}$, we know that $\textbf{probe}(W_{1,i_1+1,r}) = \textbf{true}$. This shows that the set in (2) over which the minimum is taken is non-empty, so that ω_1 is well defined. The same observation holds for $i_2, i_3, \ldots, F-1$: each ω_i

is well-defined. Now we know that $W_{i_{F-1}+1,i_F+1,r} = W_{opt}$ for some r, showing that $\omega_F = W_{opt}$. Since **probe**$(\omega_j) =$ **true** for all $j = 1, \ldots, N$ it follows that $W_{opt} = \min_{1 \leq j \leq N}\{\omega_j\}$. With these definitions, Lemma 1 applies to this problem as well.

The cost of finding an optimal row partition is basically the same as 1D partitioning with a factor of N included to account for the N binary searches each **probe** call. There are also N times as many **probe** calls needed to identify each ω_j. The overall time cost of optimally partitioning the columns is thus $O(nm + (NM \log m)^2)$. It should be noted that the one-dimensional chains-on-chains solution in [5] is easily adapted to the optimal conditional partitioning problem. The adaptation must too suffer an $O(nm)$ startup cost, plus an additional factor of N, yielding an $O(nm + NMm)$ algorithm It is also possible to ensure that the algorithm finds the "greedy" optimum, ie., the same one the probe-based algorithm finds. Lemma 2 (using $W = W_{opt}$) thus applies to this problem as well. As we will see, this implies that the dynamic-programming based solution can be used in the iterative refinement algorithm to be presented in §4.4.

4.3 Optimal 2D Partitioning

It is possible, if unpleasant, to find the optimal 2D rectilinear partitioning using the procedure just described. There are $\Omega(n^{(N-1)})$ ways of choosing a row partition; for each we can determine the optimal column partition, and thereby determine the overall optimal partitioning. It may be possible to reduce the complexity somewhat using a branch-and-bound technique to limit the number of row partitions considered, nevertheless this algorithm is exponentially complex in N. We do not yet know if a polynomial-time algorithm exists for this problem, or whether optimal 2D rectilinear partitioning is NP-complete. We do know that in practice N will be too large for us to consider this approach. In any event, a well-chosen partition will likely be adequate, even if suboptimal. Thus, we next turn our attention to a relatively fast heuristic.

4.4 Iterative Refinement

We may apply the conditionally optimal partitioning algorithm in an iterative fashion. Suppose that a row partition R_1 is given. For example, we might construct an initial row partition as follows: sum the weights of all work pieces in a common row, to create a super-piece representing that row. Find an optimal 1D partition of those super-pieces onto N processors. Use this partition as R_1, assume it to be fixed, and let C_1 be the optimal column partition, given R_1. Let $\pi_1 = \pi(R_1, C_1)$ be the cost of that partitioning. Next, fix the column partition as C_1, and let R_2 be the optimal row partitioning, given C_1. Let $\pi_2 = \pi(R_2, C_1)$. Clearly we may repeat this process as many times as we like; observe that odd iterations compute column partitions and even iterations compute row partitions.

We could choose a partition vector from either "direction", that is, choose row indices in the sequence $r_1, r_2, \ldots, r_{N-1}$ or in the sequence $r_{N-1}, r_{N-2}, \ldots, r_1$. We assume that the optimal conditional partitioning algorithm approaches the problem from the same direction every iteration, for both the row and column partitions.

A useful feature of the algorithm is that at each iteration, the cost of the solution is no worse than the cost at the previous iteration.

Lemma 3 *Given any initial row partition R_1, the sequence $\pi_1, \pi_2, \ldots,$ is monotone non-increasing.*

Proof: Without loss of generality, suppose that the partition produced at the end of iteration $i-1$ is a row partition R'; let C' be the column partition treated as fixed during iteration $i-1$. At iteration i we fix the row partition as R', and seek the optimal column partition. One of the possible column partitions is C', thus we know the column partition found will have cost no greater than $\pi(R', C')$. ∎

Iterative refinement defines a fixed-point computation, a fact that can be used as a termination condition, as shown in the following lemma.

Lemma 4 *For every starting row partition R_1 there exists an iteration I such that $R_j = R_I$ and $C_j = C_I$ for all $j > I$.*

Proof: We will need to refer to the elements of R_j and C_j by both position within the vector, and by the index j. We thus define

$$R_j = (0, r_1(j), r_2(j), \ldots, r_{N-1}(j), m)$$

and

$$C_j = (0, c_1(j), c_2(j), \ldots, c_{N-1}(j), n).$$

By Lemma 3 we know there exists an index k and a value b such that $\pi_j = b$ for all $j \geq k$. Let $j > k$, j odd. Iteration j computes column partition $C_{j/2+1}$, given fixed row partition $R_{j/2+1}$. As we compute $R_{j/2+2}$ in iteration $j+1$, a feasible partitioning is $R_{j/2+2} = R_{j/2+1}$. However, $R_{j/2+2}$ is "greedy" with respect to $C_{j/2+1}$ and b, while $R_{j/2+1}$ need not be. Thus, by Lemma 2 we must have $r_i(j/2+2) \geq r_i(j/2+1)$ for all $i = 1, 2, \ldots, M-1$. The same argument can be applied to show that $c_i(j/2+2) \geq c_i(j/2+1)$ for all $i = 1, 2, \ldots, N-1$. Since these indices can not grow without bound, eventually the row and column partitions must stop changing. ∎

Lemma 4 shows that a safe termination procedure is to iterate until the row and column partitions stop changing. It is natural to ask how many iterations are required to achieve convergence. We can bound this number, although only loosely.

Lemma 5 *Let U be the number of unique bottleneck constraints. The iterative refinement algorithm converges in $O(U \cdot (n+m))$ iterations.*

Proof: The proof of Lemma 4 implies that when convergence has not yet been achieved, no more than $n+m$ successive iterations may occur without the partition cost decreasing. The present lemma's conclusion follows from the observation that there are no more than U possible values for

$$\begin{array}{|ccc|ccc|}
\hline
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
\qquad
\begin{array}{|cc|cccc|}
\hline
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}$$

Converged Suboptimal Solution   Optimal Solution

Figure 2: Example showing that iterative refinement may converge to a suboptimal solution

the partition cost. In the worst case $U = O((nm)^2)$, since every bottleneck constraint is defined by two row indices, and two column indices. ∎

Despite the $O((nm)^2(n + m))$ bound, our experience has been that convergence is achieved in far fewer iterations, perhaps in $O(\max\{N, M\})$ iterations. One possible explanation is that the solution space for the problems we study has many local minima; another is that there are strong as-yet-undiscovered theoretical reasons for the fast convergence.

The solution found by iterative refinement is locally optimal, in the sense that we are unable to reduce the partition cost by moving any set of row indices, or any set of column indices. It may, however, be possible to improve the solution by simultaneously moving a row index and a column index. This is illustrated by the example in Figure 2. It is possible for iterative refinement to converge to the partition shown with bottleneck weight 3; this cost is reduced by appropriately moving both the row and column partitions. The practical severity of this phenomenon is unclear. Should it prove to be a problem, the algorithm might be adapted to perturbation of row and column partitions simultaneously after convergence, to determine whether any improvement in the solution quality can be achieved.

The ultimate converged cost of a partition constructed via iterative refinement depends on the starting partition R_1. We have tried a number of different seemingly natural methods for computing R_1. Somewhat to our surprise, we found that the best method (marginally) is to generate several initial partitions randomly, and keep the best resulting partition. This certainly makes sense if the partition solution space has but a few local minima. Randomly generation increases the likelihood of hitting an initial partition that leads to the optimal solution.

In the subsection to follow we discuss an application of iterative refinement to irregular mesh problems. We find that iterative refinement can effectively reduce communication costs and sometimes achieve better performance than other partitioning methods.

4.5 Application of Iterative Refinement

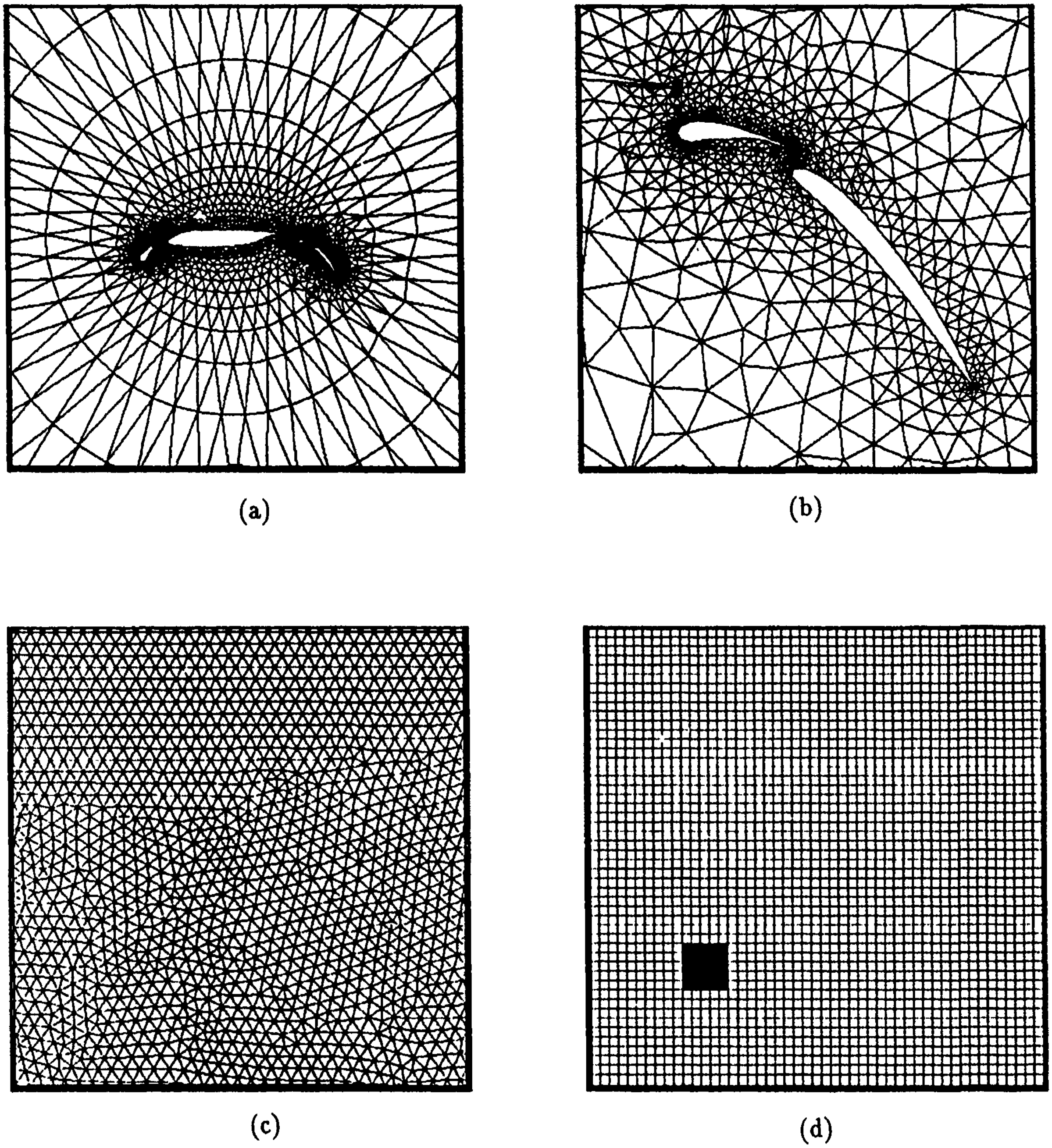
We have applied iterative refinement to irregular two-dimensional meshes typical of those used to solve two-dimensional fluid flow problems with irregular meshes. One class of mesh is "unstructured"; Figure 3(a) illustrates an unstructured grid (called Grid A henceforth) surrounding the cross-section of an air-foil [15]; (b) shows a closeup of a dense region of A. Figure 3(c) illustrates part of another unstructured grid [24], called Grid B, but one that is far less irregular. Finally, Figure 3(d) illustrates a grid C that is highly regular, except for an irregularly placed region of extremely high density. All edges in the latter grid have either vertical or horizontal orientation. As we will see, the latter type of grid gives rectilinear partitioning its greatest advantage over other techniques.

The grids we study have tens of thousands of grid points; A has 11143 points and 32818 edges, B has 19155 points and 56895 edges, C has 45625 points and 90700 edges. We chose to partition with the highest possible refinement; however, the number of grid points precludes the actual construction of a load matrix where every element represents at most one point. Instead, prior to an iteration, we construct a load matrix with either N or M rows (depending on whether we are performing a column or a row iteration), and T columns, T being the number of points. This is accomplished in time proportional to the size of this matrix. While the cost of an iteration may become dominated asymptotically by this setup cost, in our experience it makes little sense to create and store an immense, sparse matrix. On the grids described here, the complete rectilinear partitioning algorithm ran in under one minute on a Sparc 1+ workstation. The other methods were not much faster, as the I/O time for loading the grid tended to dominate them all. One exception to this occurred on partitioning the largest grid for the largest processor array. The partitioning algorithm no longer ran in memory, and suffered from a great deal of paging traffic as a consequence.

We report on experiments conducted on the three forementioned grids, using three different partitioning methods: iterative refinement, binary dissection [1], and "jagged" rectilinear partitioning [20]. Binary dissection is a commonly used technique which very carefully balances workload; however. its partitions are constructed without regard for communication patterns. Jagged rectilinear partitioning has recently been proposed to overcome some of binary dissection's problems. The domain is first divided in N strips, of approximately equal weight. Following this, each strip is individually divided into M rectangles of approximately equal weight. While partition cuts do span the entire domain in one dimension, they are "jagged" in the other.

We also experimented with so-called "strip" partitions, defined by the optimal 1D solution of the projection of these 2D problems onto the line. We do not report the results of these experiments, as strip partitions were uniformly worse than the ones we study, due primarily to excessive interprocessor communication (even if primarily local), caused by a poor area to perimeter ratio.

Our experiments assessed the overall cost of a partition to a processor to be the sum of the weights of the grid points it is assigned, plus a communication cost that depends on both the architecture, and the mapping. Computations on grids of this type are based primarily on edges; hence the cost of a grid point is taken to be the total number of its edges. The communication



(a)

(b)

(c)

(d)

Figure 3: Grids used in application study. (a) is a highly unstructured mesh around an airfoil cross-section, (b) is a closeup. (c) is a more regular unstructured mesh; (d) is an artificial mesh with perfectly orthogonal edges and an offset region of high density.

cost is defined by edges that span different processors. Each edge is classified as being internal, local, or global, depending on whether the edge is completely contained in one processor, spans processors which are adjacent in the processor mesh, or spans processors which are more distantly separated. In the experiments we present, "adjacent" means adjacent in a North-East-West-South mesh. We comment later on results obtained assuming a mesh that includes direct connections between diagonally adjacent processors as well.

Each processor's local communication cost is taken to be the number of its local edges; the global communication cost is the number of global edges times a parameter G. An edge's communication cost is charged to both of its processors. The cost of a partition is the maximum cost of any processor in that partition. We may estimate speedup as the sum of the weights of all grid points divided by the maximum processor cost. We have experimentally compared a number of these estimates with actual speedup measurements on an Intel iPSC/860, and found them to be reasonably accurate. Of course, the iPSC/860 does not have the same type of local/global communication differential as that assumed here; the cost of a message is largely insenstive to the distance it must travel (at least in the absence of serious network conjestion). Nevertheless it seems intuitive that scaling global communication by a parameter G is a appropriate model for the architectures of interest.

We use three metrics to characterize a partition. One is f_I, the fraction of edges that are internal; another is f_L the fraction of external edges that are local. Finally, f_U is the average processor workload divided by the load of the maximally loaded processor, under the assumption that all communication has cost 0. f_I and f_L are measures of how well the partition preserves locality of communication, while f_U is a measure of how well the partition balances workload. Table 1 presents these quantities, measured on our problem set, mapping to 16 × 16, 32 × 32, and 64 × 64 processor arrays. For both f_I and f_E we see that rectilinear partitioning is somewhat better at keeping edges internal, and that it excels at keeping external edges local. The price it pays for this locality is increased load imbalance, as is evident from the f_U values. Of course, this is to be expected, since a rectilinear partition is a constrained version of a binary dissection.

Figures 4, 5, and 6, give estimated processor efficiencies on the three grids, measured as the estimated speedup divided by the number of processors. Each performance curve is parameterized by G, in order to show how performance is affected by an increasing cost differential between local and global communication. Each graph plots performance curves for each of the three partitioning methods (encoded here as BD,JP,and RP) with 16 × 16, 32 × 32, and 64 × 64 processor arrays. All initial RP row partitions were selecting by computing the optimal 1D partition for N processors.

For grid A we see that BD has a clear advantage over the other methods when global communication is as cheap as local. However, as G grows it increasingly suffers from its global edges; On the 16 × 16 and 32 × 32 arrays JP surpasses it once $G > 3$: however it fails to surpass BD at all on the 64 × 64 array. On a 16 × 16 array, RP surpass BD once $G \geq 5$, and surpasses JP once $G \geq 9$. On the 32 × 32 array RP surpass both BD and JP after $G \geq 5$, whereas on the 64 × 64 array it is bested by both BD and JP. At this extreme point most edges go off processor, and the workload is small. BD's advantage in load balancing then dominates. Observe however that performance at the right end of the curve is not good; this may be indicative of placing too small a problem on the

Processor array	16 × 16	32 × 32	64 × 64
	(f_I, f_E, f_U)	(f_I, f_E, f_U)	(f_I, f_E, f_U)
Binary Dissection	(0.73, 0.32, 0.98)	(0.49, 0.29, 0.92)	(0.19, 0.24, 0.72)
Jagged Partitioning	(0.70, 0.79, 0.84)	(0.45, 0.73, 0.68)	(0.19, 0.52, 0.53)
Rectilinear Partitioning	(0.77, 0.91, 0.27)	(0.61, 0.82, 0.27)	(0.37, 0.68, 0.24)

f_I, f_E, and f_U values for Grid A

Processor array	16 × 16	32 × 32	64 × 64
	(f_I, f_E, f_U)	(f_I, f_E, f_U)	(f_I, f_E, f_U)
Binary Dissection	(0.84, 0.37, 0.98)	(0.67, 0.37, 0.96)	(0.37, 0.38, 0.86)
Jagged Partitioning	(0.84, 0.95, 0.98)	(0.67, 0.87, 0.95)	(0.39, 0.77, 0.86)
Rectilinear Partitioning	(0.84, 0.97, 0.92)	(0.68, 0.94, 0.80)	(0.44, 0.86, 0.66)

f_I, f_E, and f_U values for Grid B

Processor array	16 × 16	32 × 32	64 × 64
	(f_I, f_E, f_U)	(f_I, f_E, f_U)	(f_I, f_E, f_U)
Binary Dissection	(0.91, 0.27, 0.99)	(0.82, 0.29, 0.98)	(0.62, 0.30, 0.92)
Jagged Partitioning	(0.91, 0.92, 0.98)	(0.82, 0.76, 0.98)	(0.64, 0.66, 0.92)
Rectilinear Partitioning	(0.91, 1.00, 0.85)	(0.83, 1.00, 0.85)	(0.70, 1.00, 0.69)

f_I, f_E, and f_U values for Grid C

Table 1: Fraction f_I of internal edges, fraction f_L of external edges which are local, and processor utilization f_U under no communication costs, for different meshes, processor arrays, and partitioning methods

machine.

Grid B is much more regular than A, a fact that translates into higher performance under higher values of G. On the two smaller arrays the RP curves cross the JP and BD curves in the region of $G = 5$. On the largest array JP is somewhat better than the other methods, while the RP and BD curves are surprisingly similar after $G > 3$.

Grid C was constructed specifically to highlight RP's advantages over the other methods. Under RP, none of its edges are global, so performance is insensitive to G. RP's cross-over points are again in the region $G \in (3, 5)$; owing to its complete avoidance of global costs, its performance is substantially better than the others under high values of G.

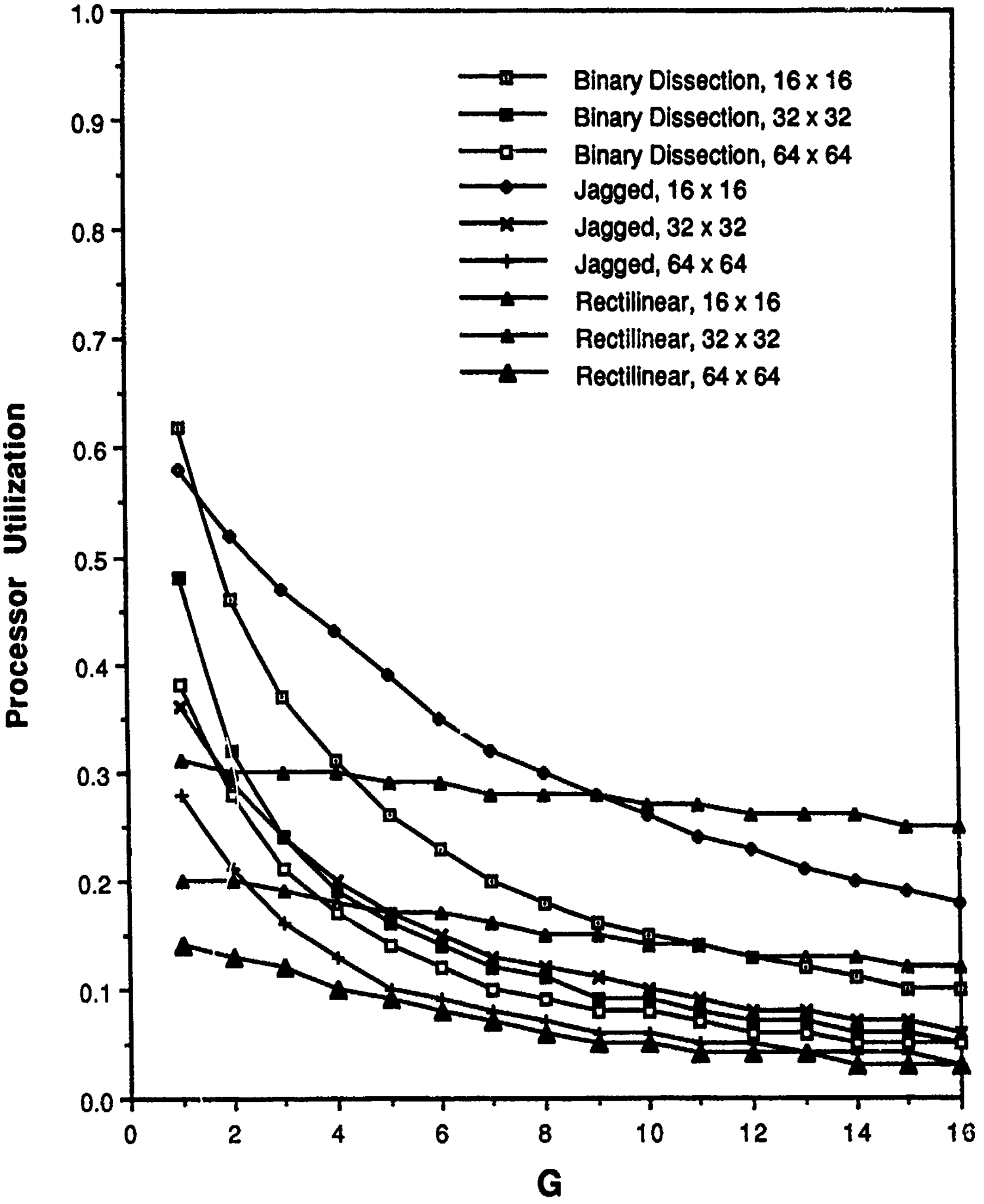


Figure 4: Processor utilizations on the BD, JP, and RP partitions of grid A, for 16 × 16, 32 × 32, and 64 × 64 processor arrays

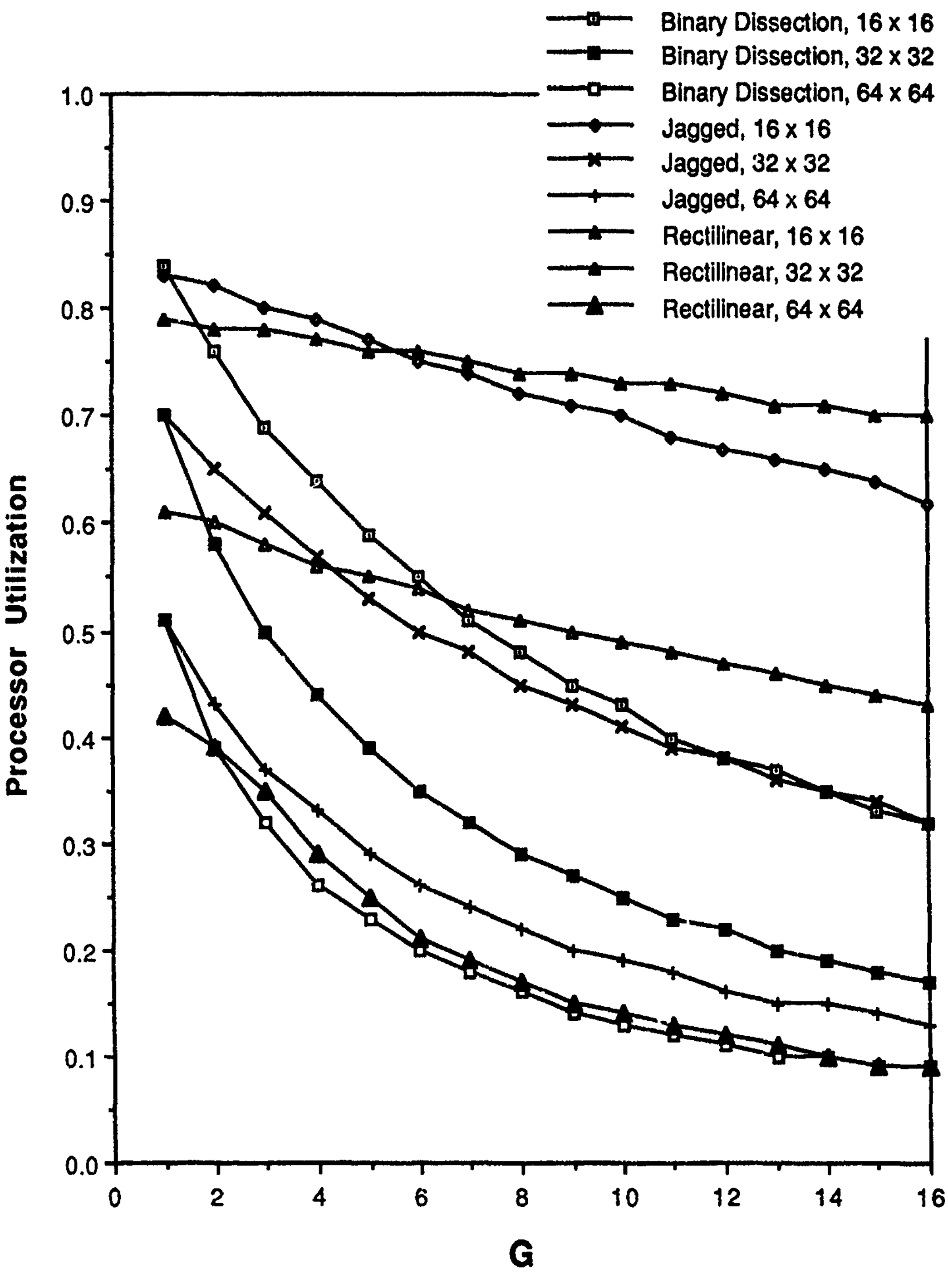


Figure 5: Processor utilizations on the BD, JP, and RP partitions of grid B, for 16 x 16, 32 x 32, and 64 x 64 processor arrays

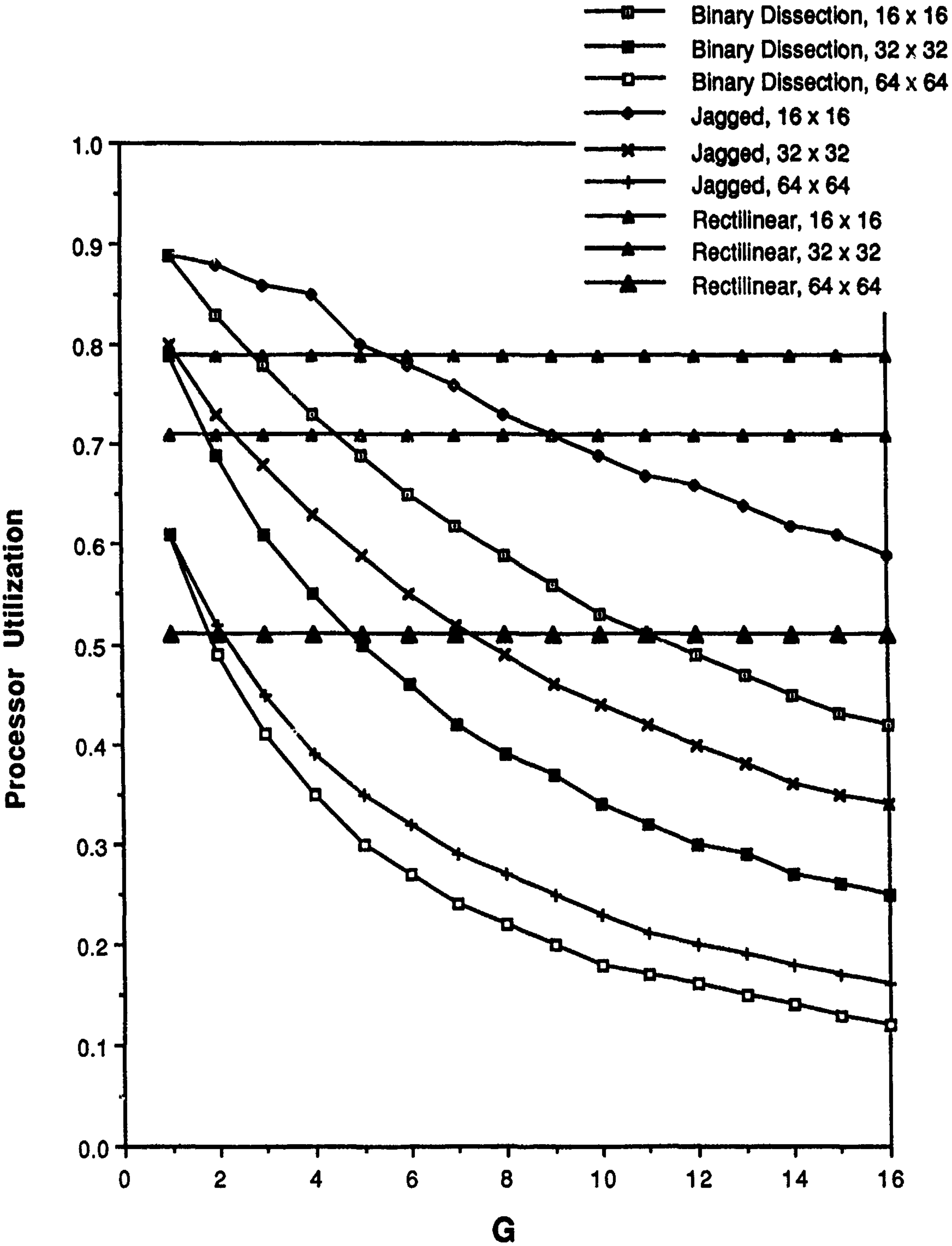


Figure 6: Processor utilizations on the BD, JP, and RP partitions of grid C, for 16 x 16, 32 x 32, and 64 x 64 processor arrays

Processor array	16×16	32×32	64×64
Grid A	13	11	37
Grid B	9	11	19
Grid C	5	5	5

Table 2: Iterations used by iterative refinement to converge

On these problems, rectilinear partitioning required far fewer iterations to reach convergence than would be suggested by Lemma 5. Table 2 gives the number of iterations required for each of the nine rectilinear partitions generated.

We also evaluated the cost of these partitions assuming that diagonally adjacent processors are connected in the local network. In every case the performance of BP was completely unaffected. RP's performance improved slightly, usually by no more than 10%. JP's performance improved sharply, to the extent that it outperforms RP on almost all the Grid A and Grid B partitions. RP retains its superiority on Grid C. These results suggest that jagged partitions effectively capture locality when that locality is defined to include diagonally connected processors. Of course, there is no guarantee that a jagged partition will map perfectly onto an 8-neighbor mesh; an interesting future line of inquiry is to develop algorithms that guarantee such locality. Rectilinear partitions are most desirable when the rectilinear constraint matches the rectilinear nature of North-East-West-South meshes.

The data presented here indicates that rectilinear partitions have their utility. When global communication values are high, it is worthwhile to accept some load imbalance for the sake of communication locality. On the other hand, it is clear that rectilinear partitions are not desirable when the problem is highly irregular and global communication is comparatively cheap. We plan further experimentation with these partitioning strategies on actual codes on actual machines.

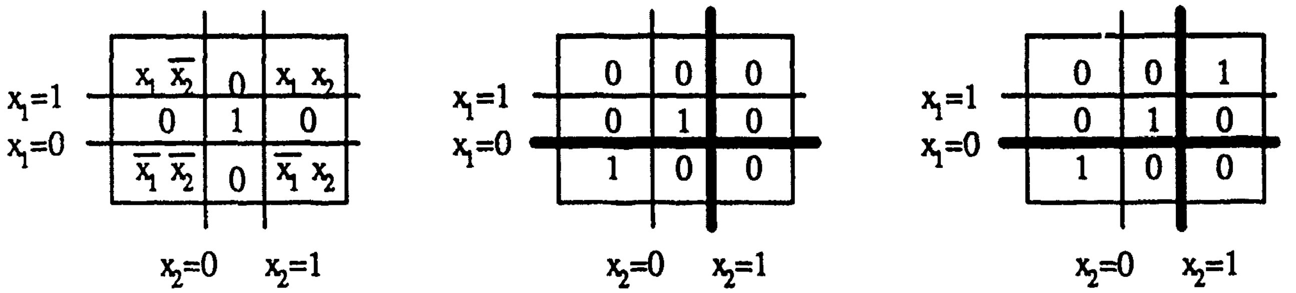
5 Three Dimensional Partitioning

We have already seen that RPP in one dimension can be solved in polynomial time; it is not yet known whether the two-dimensional problem is tractable. In this section we demonstrate that RPP in three dimensions is NP-complete. We establish the fact by demonstrating that an arbitrary monotone 3SAT problem [8] can be solved by any three-dimensional RPP algorithm. Since the monotone 3SAT problem is NP-complete, so is RPP in three dimensions.

The general 3SAT problem has the following form. We are given n Boolean literals $x_1, \ldots, x_n$, and m clauses $C_1, \ldots, C_m$. Each clause is the disjunction of three distinct literals, each of which may be complimented or uncomplimented. For example, $(x_1 + \bar{x}_3 + x_{17})$ and $(\bar{x}_1 + \bar{x}_2 + x_{14})$ are two clauses. The 3SAT problem is to find a Boolean assignment for each literal such that every clause evaluates to `true`. The *monotone* 3SAT problem requires that every given clause have either all complimented literals, or all uncomplimented literals. A useful consequence of the monotone restriction is that for any given triple of literals (x_i, x_j, x_k) there are at most two clauses involving all three simultaneously—one where they are all complimented, and one where they are not. It has been shown that the monotone 3SAT problem is NP-completé [8]. Minor modifications to the approach we develop will work for general 3SAT problems; it is simply easier to describe the transformation if we assume the clauses are monotone.

A choice of partitioning can be interpreted as an assignment of literal values and assessment of a clause's truth value. We first introduce these ideas by application to the monotone 2SAT problem, where clauses have two literals (2SAT can be solved in polynomial time). Let x_1 and x_2 be two literals; only two monotone clauses are possible, $(x_1 + x_2)$ or $(\bar{x}_1 + \bar{x}_2)$. In either case, only one assignment of values to the literals can cause the clause not to be satisfied, $x_1 = x_2 = 0$ in the former case and $x_1 = x_2 = 1$ in the latter. We capture this in a partitioning framework with a 3×3 domain with binary workload weights, to be partitioned into four pieces. The center weight is 1; one corner is also weighted with 1 depending on the clause, and all other weights are 0. Figure 7(a) illustrates the domain, and the assignment of *infeasibility products* x_1x_2, $x_1\bar{x}_2$, $\bar{x}_1x_2$, and $\bar{x}_1\bar{x}_2$ to opposing corners. The choice of a row partition corresponds to an assignment to x_1, the choice of a column partition corresponds to an assignment to x_2. Our weighting rule is to assign a 1 to a corner whose infeasibility product is true when the corresponding truth assignment fails to satisfy the clause. Thus, if $(x_1 + x_2)$ is a problem clause, then the $\bar{x}_1\bar{x}_2$ corner is given a 1; if $(\bar{x}_1 + \bar{x}_2)$ is a clause then the x_1x_2 corner is given a 1. If both clauses appear in the problem, both corresponding corners are weighted by 1. This is equivalent to requiring that $x_1 \oplus x_2 = 1$ ($\oplus$ being the exclusive OR operator). Also, in our problem transformation it will be possible for x_1 and x_2 to represent the same literal. If this is the case, we place 1s in the $x_1\bar{x_2}$ and $\bar{x_1}x_2$ corners, in order to force a common selection for the literal, in both its column and row representations. Figure 7(b) and (c) illustrates the weighting corresponding to conditions $(x_1 + x_2)$ and $(x_1 \oplus x_2)$ respectively, and shows the partition corresponding to the assignment $x_1 = 0$, $x_2 = 1$. Observe that the bottleneck weight is 1, whereas it would be 2 if the infeasible assignment $x_1 = 0$ and $x_2 = 0$ were chosen. The infeasible assignment is the only one achieving a bottleneck cost of 2. This is true of the construction for any clause, and is the key to determining whether the assignment corresponding to some partitioning satisfies all clauses.

A monotone 2SAT problem can be transformed into a rectilinear partitioning problem using the ideas expressed above. Given n literals $x_1, \ldots, x_n$ we will create a $(4n-1) \times (4n-1)$ binary domain. For each variable we assign three contiguous rows, and three contiguous columns. Variables' sets of rows and columns are separated by a single "padding" row and single "padding" column whose purpose will be to force a partition within each variable's set of rows, and within each variable's set of columns. We assign 1s and 0s described above for the 3×3 intersection of x_i's rows and x_j's columns. Elements at the intersection of two variables that never appear in the same clause are all assigned value 0. We place a 0 wherever a "middle" row for a variable meets a padding column; likewise, we place a 0 wherever a variable's middle column meets a padding row. Otherwise, every other entry of a padding row or a padding column is 1. The construction for the problem



(a) General construction of domain to represent a clause

(b) Domain for x_1+x_2, partition for assignment $x_1=0$, $x_2=1$

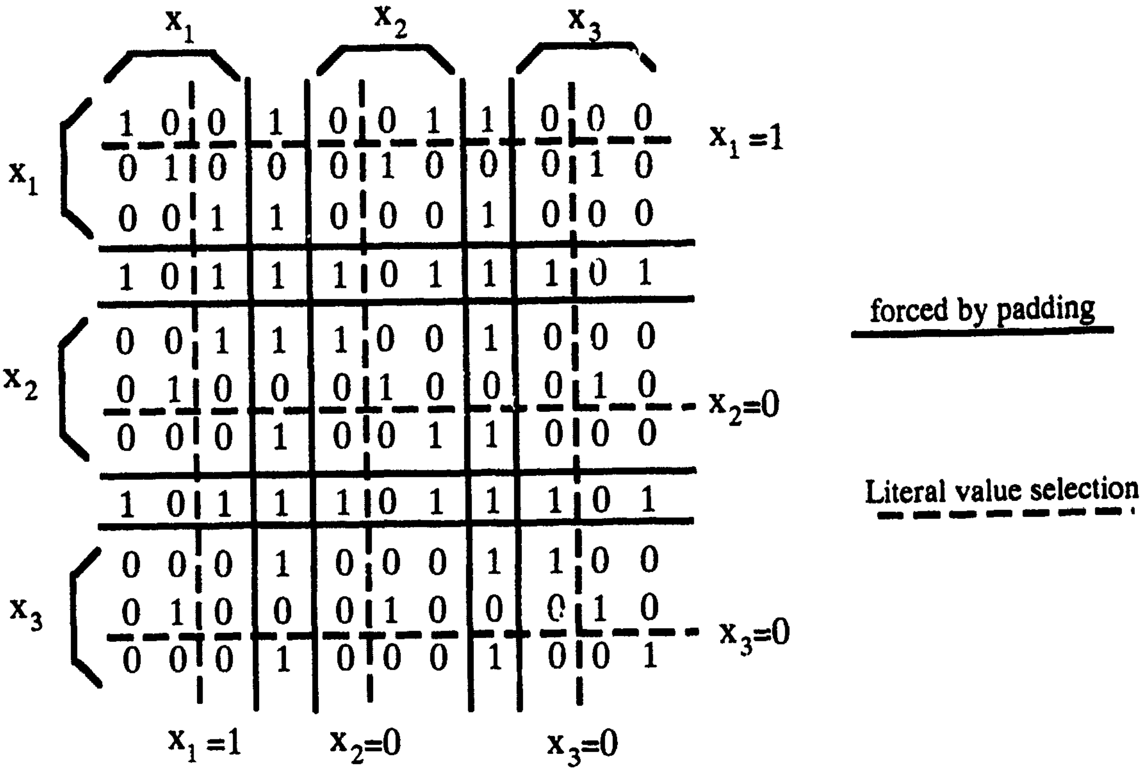
(c) Domain for $x_1 \oplus x_2$

Figure 7: Transformation of 2SAT Problem into Rectilinear Partitioning Problem

$(x_1 + x_2)(\bar{x}_2 + \bar{x}_3)$ is shown in Figure 8. We seek an optimal rectilinear partitioning of this domain onto a $(3n - 1) \times (3n - 1)$ array of prcessors. Weights in padding rows and columns are defined in such a way that for a bottleneck weight of 1 to be achieved it is necessary that a partition never group padding and non-padding rows or group padding and non-padding columns. This forces a partition of every variable's rows, and every variable's columns.

If the domain can be partitioned and achieve a bottleneck cost of 1, then the 2SAT problem is solved by the assignment implicit in the optimal partitioning. Otherwise the 2SAT problem cannot be solved. Figure 8 also illustrates the partition corresponding to the solution $x_1 = 1$, $x_2 = 0$, and $x_3 = 0$.

The extension of these constructs to three dimensions is straightforward. Let x_1, x_2, x_3 be literals. In a monotone 3SAT problem the only possible clauses are $(x_1+x_2+x_3)$ and $(\bar{x}_1+\bar{x}_2+\bar{x}_3)$; in the former case only the assignment $x_1 = x_2 = x_3 = 0$ fails to satisfy the clause, in the latter case only $x_1 = x_2 = x_3 = 1$ fails to satisfy the clause. In the event that both clauses appear, their conjunction is not satisfied it and only if the variables are all assigned the same value. Now let us associate a $3 \times 3 \times 3$ *clause region* with these literals. The 2SAT construction associated x_1 with the Y dimension and x_2 with the X dimension; we augment this and associate x_3 with the Z dimension. It is convenient to view a clause region as three stacked 3×3 arrays with XY orientation. The centermost element of the middle array will have value 1, all other elements of the middle array are 0. Like the 2SAT problem, the four corners of the lowest 3×3 array represent products of all three literals. In the 3SAT case, all products in the "bottom" array include $\bar{x}_3$ and all products in



1	0	0	1	0	0	1	1	0	0	0
0	1	0	0	0	1	0	0	0	1	0
0	0	1	1	0	0	0	1	0	0	0
1	0	1	1	1	0	1	1	1	0	1
0	0	1	1	1	0	0	1	0	0	0
0	1	0	0	0	1	0	0	0	1	0
0	0	0	1	0	0	1	1	0	0	0
1	0	1	1	1	0	1	1	1	0	1
0	0	0	1	0	0	0	1	1	0	0
0	1	0	0	0	1	0	0	0	1	0
0	0	0	1	0	0	0	1	0	0	1

Figure 8: Example of 2SAT problem $(x_1 + x_2)(\bar{x}_2 + \bar{x}_3)$ mapped to 2D rectilinear partitioning of 9×9 binary domain onto 6×6 array of processors. Partition of solution $x_1 = 1, x_2 = 0, x_3 = 0$ is shown.

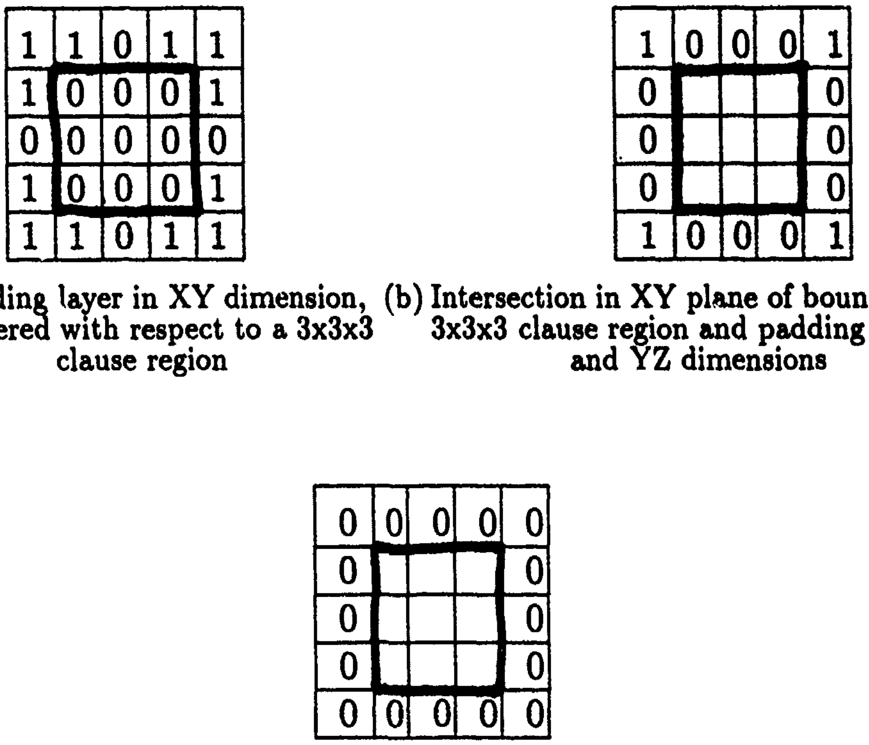
the "top" array include x_3. The x_1 and x_2 combinations are identical to the 2SAT problem. For example, the infeasibility products in the northwest, northeast, southwest, and southeast corners of the top array are $\bar{x}_1 x_2 x_3$, $\bar{x}_1 \bar{x}_2 x_3$, $x_1 x_2 x_3$, and $x_1 \bar{x}_2 x_3$ respectively. Like the 2SAT problem, we weight a corner with 1 if the truth assignment satisfying the corresponding infeasibility product fails to satisfy the clause. Thus, if $(x_1 + x_2 + x_3)$ appears as a clause, we place a 1 in the $\bar{x}_1 \bar{x}_2 \bar{x}_3$ corner. If $(\bar{x}_1 + \bar{x}_2 + \bar{x}_3)$ is a clause then we place a 1 in the $x_1 x_2 x_3$ corner. Both 1s are placed if both of these clauses appear in the problem. All other entries of the clause region are 0. All clause regions corresponding to three distinct literals that do not appear in a clause are zeroed out. Clause regions involving intersections of a literal and itself are weighted to ensure that a bottleneck value of 1 is achieved only if partitions are chosen corresponding to the same selection of literal value in each dimension. For example, if x_1 and x_3 happen to be the same literal, then a 1 is placed in any corner whose product involves $x_1 \bar{x}_3$ or $\bar{x}_1 x_3$.

Assignment of a value for x_1 corresponds to selection of a plane with YZ orientation. The plane's intersection with each layer in the clause region looks the same—it is either the $x_1 = 0$ line or the $x_1 = 1$ line as seen in the 2SAT problem (Figure 7). Similarly, assignment of a value for x_2 corresponds to a plane whose intersection with each layer is identical, either the line for $x_2 = 0$ or the line for $x_2 = 1$. Finally, selection of $x_3 = 1$ is accomplished by selecting an XY plane that separates the bottom two layers from the top layer, while selection of $x_3 = 0$ separates the bottom layer from the top two. Under this construction, selection of planes corresponding to an assignment that makes an infeasibility product true will place the centermost 1 in the same volume as the "infeasibility 1", giving rise to a bottleneck weight of 2. This fact is important enough to state formally.

Lemma 6 *Let $I(x_1, x_2, x_3)$ be any infeasibility product whose position in a clause region is set to value 1. Then any partition whose associated assignment sets $I(x_1, x_2, x_3) = 1$ places the infeasibility 1, and the clause region's center 1 in the same partition volume. The bottleneck cost of any such partition is at least 2.*

Like the 2SAT mapping, we add "padding" layers to ensure that any partition with cost 1 must choose one of two planes in each dimension of each clause region. The assignment of 1s and 0s to padding layers is similar to the 2SAT case. Figure 9 defines the assignment in terms of how each layer's elements are weighted in the immediate vicinity of a clause region. Figure 9(a) shows how a portion of the padding layer with (XY orientation) is weighted when centered directly above or below a clause region (the heavy lines illustrate how the clause region is positioned). The only way to separate the three 1s in each corner is to choose the four partitioning layers with XZ orientation and four with YZ orientation that do not intersect the 3 × 3 core. These layers ensure that no elements on the XZ and YZ faces of the clause region will be grouped with any elements from any other clause region—at least if a bottleneck cost of 1 is to be achieved. Figures 9(b) and (c) then show how to weight elements in padding layers with XZ and YZ orientation, depending on whether the padding layer intersects a layer containing a boundary or middle layer of the clause region. Weights for the clause region (which is outlined) are not included. The corner 1s seen in Figure 9(b) are adjacent to the corner 1s seen in Figure 9(a); in order to achieve a bottleneck cost of 1 it will be place two partitioning planes with XY orientation to contain the XY padding layer. This ensures that any element at an XY face of the clause region will not be grouped with elements from any other clause region.

To transform a monotone 3SAT problem we construct a $(4n - 1) \times (4n - 1) \times (4n - 1)$ domain. The first three coordinate positions in each dimension correspond to x_1; the fourth coordinate position in each dimension corresponds to padding, the next three coordinate positions in each dimension correspond to x_2, and so on. The domain is weighted as described above. We have seen that in order to achieve a bottleneck cost of 1 it is necessary to contain each padding layer with two partitioning planes. This defines $(2n - 1)$ planes orthogonal to each dimension. Furthermore, it is also necessary to appropriately partition each clause region in each dimension. This leads to an additional n partitioning planes orthogonal to each dimension. Consequently, the dimensions of the target architecture are $(3n - 1) \times (3n - 1) \times (3n - 1)$.



1	1	0	1	1
1	0	0	0	1
0	0	0	0	0
1	0	0	0	1
1	1	0	1	1

(a) Padding layer in XY dimension, centered with respect to a 3x3x3 clause region

1	0	0	0	1
0				0
0				0
0				0
1	0	0	0	1

(b) Intersection in XY plane of boundary layer for 3x3x3 clause region and padding layers in XZ and YZ dimensions

0	0	0	0	0
0				0
0				0
0				0
0	0	0	0	0

(c) Intersection in XY plane of middle layer for 3x3x3 clause region and padding layers in XZ and YZ dimensions

Figure 9: Weighting of elements in padding planes

Finally, we must show that under this construction, the 3SAT problem has a solution if and only the corresponding three dimensional partitioning problem achieves a cost of 1. This is an easy consequence of the fact that each volume in a partitioned clause region has weight no greater than 1 if and only if no clause region is partitioned to satisfy one of its infeasibility conditions (either clause infeasibility or conflicting assignment of the same literal). Since monotone 3SAT is in NP, then three dimensional RPP is in NP. Since one can always check in polynomial time whether a proposed RPP solution achieves bottleneck cost 1, three-dimensional RPP is NP-complete. In fact, since the RPP matrix we construct is binary, we have a stronger result.

Theorem 7 *Binary RPP in three dimensions is NP-complete.*

6 Summary

This paper examines the problem of partitioning with one, two, or three dimensional rectilinear partitions. When used to balance workload in data parallel computations having localized communication, such partitions can be expected to reduce the need for expensive global communication.

For the one-dimensional case we improved upon the best published solution to date when $m \gg M$, reducing the cost of finding the optimal partition of m modules among M processors to $O(m + (M \log m)^2)$. For the two-dimensional case we showed how it is possible to find the best possible partitioning in a given dimension, provided that the partition in the alternate dimension remains fixed. This result can be used to find the optimal partition in two dimensions, but with exponentially large cost (if the numbers of processors in both dimensions is a problem parameter). The result also serves as the basis for a heuristic that iteratively improves upon a solution. The heuristic is shown to converge to a fixed point, in a bounded number of iterations. Empirical studies show that the heuristic may provide some performance advantage when the differential between the local and global network bandwidth is moderately large. Finally, we showed that the problem of finding an optimal three dimensional rectilinear partition is NP-complete.

Acknowledgements

We thank Adam Rifkin for his help in programming. Discussions on this problem with Joel Saltz and Shahid Bokhari were, as always, quite useful.

References

[1] M.J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. on Computers*, C-36(5):570–580, May 1987.

[2] F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, 4:439–458, 1987.

[3] S. H. Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers*, 37(1):48–57, January 1988.

[4] M.Y. Chan and F.Y.L. Chin. On embedding rectangular grids in hypercubes. *IEEE Trans. on Computers*, 37(10):1285–1288, October 1988.

[5] H.-A. Choi and B. Narahari. Algorithms for mapping and partitioning chain structured parallel computations. In *Proceedings of the 1991 Int'l Conference on Parallel Processing*, St. Charles, Illinois, August 1991. To appear.

[6] F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hyercube by recursive mincut partitioning. *Journal of Parallel and Distributed Computing*, 10:35–44, 1990.

[7] G. Fox, A. Kolawa, and R. Williams. The implementation of a dynamic load balancer. Technical Report C3P-287a, Caltech Report, February 1987.

[8] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman and Co., New York, 1979.

[9] C.-T. Ho and S.L. Johnsson. On the embedding of arbitrary meshes in boolean cubes with expansion two dilation two. In *Proceedings of the 1987 Int'l Conference on Parallel Processing,* pages 188–191, August 1987.

[10] O.H. Ibarra and S.M. Sohn. On mapping systolic algorithms onto the hypercube. *IEEE Trans. on Parallel and Distributed Systems,* 1(1):48–63, January 1990.

[11] M.A. Iqbal. Approximate algorithms for partitioning and assignment problems. Technical Report 86-40, ICASE, June 1986.

[12] M.A. Iqbal and S.H. Bokhari. Efficient algorithms for a class of partitioning problems. Technical Report 90-49, ICASE, July 1990.

[13] R. Kincaid, D.M. Nicol, D Shier, and D. Richards. A multistage linear array assignment problem. *Operations Research,* 38(6):993–1005, 1990.

[14] C.-T. King, W.-H. Chou, and L.M. Ni. Pipelined data-parallel algorithms. *IEEE Trans. on Parallel and Distributed Systems,* 1(4):470–499, October 1990.

[15] D.J. Mavriplis. Multigrid solution of the two-dimensional Euler equations on unstructured triangular meshes. *AIAA Journal,* 26:824–831, 1988.

[16] R.G. Melhem and G.-Y. Hwang. Embedding rectangular grids into square grids with dilation two. *IEEE Trans. on Computers,* 39(12):1446–1455, Decemeber 1990.

[17] D.M. Nicol and D.R. O'Hallaron. Improved algorithms for mapping parallel and pipelined computations. *IEEE Trans. on Computers,* 40(3):295–306, 1991.

[18] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Trans. on Computers,* C-36(7):845–858, July 1987.

[19] P. Sadayappan and F. Ercal. Nearest-neighbor mapping of finite element graphs' onto processor meshes. *IEEE Trans. on Computers,* 36(12):1408–1424, December 1987.

[20] J. Saltz, S. Petiton, H. Berryman, and A. Rifkin. Performance effects of irregular communication patterns on massively parallel multiprocessors. *Journal of Parallel and Distributed Computing,* 1991. To appear. Available as ICASE Report 91-12, ICASE, NASA Langley Research Center, MS 132C, Hampton, VA 23665.

[21] D.S. Scott and R. Brandenburg. Minimal mesh embeddings in binary hypercubes. *IEEE Trans. on Computers*, 37(10):1284–1285, October 1988.

[22] L.W. Tucker and G.G. Robertson. Architecture and applications of the Connection Machine. *Computer*, 21:26–38, August 1988.

[23] S. Vavasis. Automatic domain partitioning in three dimensions. *SIAM Journal on Scientific and Statisical Computing*, July 1991. To appear.

[24] D.L. Whitaker, D.C. Slack, and R.W. Walters. Solution algorithms for the two-dimensional Euler equations on unstructured meshes. In *Proceedings AIAA 28th Aerospace Sciences Meeting*, Reno, Nevada, January 1990.

Report Documentation Page

1. Report No. NASA CR-187601 ICASE Report No. 91-55	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle

RECTILINEAR PARTITIONING OF IRREGULAR DATA PARALLEL COMPUTATIONS

5. Report Date

July 1991

6. Performing Organization Code

7. Author(s)

David M. Nicol

8. Performing Organization Report No.

91-55

10. Work Unit No.

505-90-52-01

9. Performing Organization Name and Address

Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

11. Contract or Grant No.

NAS1-18605

13. Type of Report and Period Covered

Contractor Report

12. Sponsoring Agency Name and Address

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Michael F. Card

Submitted to Journal of Parallel and Distributed Computing

16. Abstract

This paper describes new mapping algorithms for domain-oriented data-parallel computations, where the workload is distributed irregularly throughout the domain, but exhibits localized communication patterns. We consider the problem of partitioning the domain for parallel processing in such a way that the workload on the most heavily loaded processor is minimized, subject to the constraint that the partition be perfectly rectilinear. Rectilinear partitions are useful on architectures that have a fast local mesh network and a relatively slower global network; these partitions heuristically attempt to maximize the fraction of communication carried by the local network. This paper provides an improved algorithm for finding the optimal partition in one dimension, new algorithms for partitioning in two dimensions, and shows that optimal partitioning in three dimensions is NP-complete. We discuss our application of these algorithms to real problems.

17. Key Words (Suggested by Author(s))

mapping, partitioning, rectilinear, algorithms

18. Distribution Statement

61 - Computer Programming and Software

Unclassified - Unlimited

19 Security Classif. (of this report)	20 Security Classif. (of this page)	21 No of pages	22. Price
Unclassified	Unclassified	29	A03